3
Algoritmi Approssimati

Gli algoritmi di approssimazione sono algoritmi che garantiscono una soluzione sufficientemente vicina a quella ottimale. Sono utili per risolvere problemi non risolvibili con algoritmi deterministici o per migliorare l'efficienza della soluzione deterministica, il tutto a scapito della precisione della soluzione trovata. Ovviamente è richiesto che l'algoritmo di approssimazione funzioni in modo efficiente, cioè in tempo polinomiale. Quindi l'obiettivo di un algoritmo di approssimazione è quello di trovare una soluzione in tempo polinomiale offrendo una garanzia su quanto vicina all'ottimo (o lontana dall'ottimo) è la soluzione fornita.

In questo capitolo vedremo algoritmi di approssimazione per alcuni problemi difficili.

3.1 Approssimazione e efficienza

Considereremo problemi di ottimizzazione: o ricerca di un massimo (profitto) o ricerca di un minimo (costo). Un algoritmo A garantisce un'approssimazione ρ, $\rho \geq 1$, se il valore V della soluzione prodotta da A non si discosta dal valore OPT della soluzione ottima, più di un fattore ρ, cioè se

$$V \leq \rho \cdot OPT \quad \text{per problemi di minimizzazione,}$$

oppure se

$$V \geq \frac{OPT}{\rho} \quad \text{per problemi di massimizzazione,}$$

come mostrato nella Figura 3.1

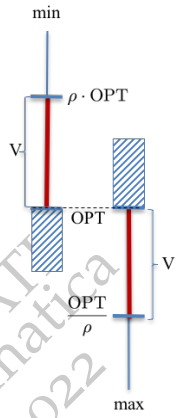

Figura 3.1: Approssimazione per problemi di minimzzazione e massimizzazione

Quando questa condizione è garantita, l'algoritmo A viene detto ρ-approssimato. Ad esempio un algoritmo 2-approssimato garantisce che la soluzione fornita non sia più grande del doppio della soluzione ottima, nel caso di problemi di minimizzazione, oppure più piccola della metà dell'ottimo nel caso di problemi di massimizzazione. Si noti come deve necessariamente essere $\rho \geq 1$. Gli algoritmi non approssimati, cioè quelli che trovano la soluzione ottima, hanno un fattore di approssimazione pari a 1, sono cioè 1-approssimati. Quanto più grande è il fattore di approssimazione tanto peggiore potrà essere la soluzione fornita dall'algoritmo di approssimazione.

Si noti come abbiamo implicitamente assunto che ρ sia una costante. In realtà il fattore di approssimazione può dipendere dalla taglia del problema o anche dal tempo

di esecuzione dell'algoritmo, come vedremo nel prosieguo del capitolo. Tuttavia, molti degli algoritmi di approssimazione che vedremo garantiscono una soluzione il cui valore si può allontanare dal valore ottimo per un fattore multiplicativo costante, e questo succede per la maggior parte dei casi.

Per il problema VertexCover vedremo algoritmi 2-approssimati. Si noti che questo significa che una soluzione può discostarsi da quella ottima fino al 100%. Ovviamente ridurre questo fattore è importante in pratica. Un algoritmo approssimato che garantisca una soluzione che si discosti molto poco da quella ottima è praticamente un algoritmo che fornisce quasi la soluzione ottima; ad esempio una soluzione che si discosta dello 0.5% da quella ottima potrebbe essere più che sufficiente[1]. In alcuni casi riusciamo a fornire degli algoritmi che possono produrre una soluzione approssimata buona quanto si vuola; ovviamente per fare ciò serve più tempo ma il tempo necessario, seppur maggiore, rimane polinomiale. Approfondiremo questo aspetto nella Sezione 3.5. Sfortunatamente questo non è sempre possibile e ci sono casi in cui gli algoritmi di approssimazione sono anche peggiori, non riuscendo a garantire un discostamento di un fattore costante e quindi la soluzione trovata peggiora al crescere della grandezza dell'input.

[1] Ovviamente la percentuale va valutata in relazione allo specifico problema.

Una motivazione può essere identificata nel seguente fatto: se $\mathcal{P} \neq \mathcal{NP}$, i problemi $\mathcal{NP}$-completi differiscono considerevolmente per quanto riguarda la loro approssimabilità. Ad esempio si può dimostrare che se $\mathcal{P} \neq \mathcal{NP}$, l'algoritmo approssimato che vedremo per il problema della selezione dei centri fornisce la migliore approssimazione possibile se ci vincoliamo ad usare algoritmi polinomiali. Per altri problemi, come ad esempio il problema del ricoprimento con vertici, l'algoritmo che vedremo è il migliore che si conosce ma dire se è il migliore possibile è un problema aperto. Non discuteremo di limiti dell'approssimabilità dei problemi: anche se per alcuni problemi (come quello della selezioni dei centri) dimostrare un limite all'approssimabilità non è molto difficile, spesso tali dimostrazioni sono estremamente tecniche.

3.2 VertexCover

Consideriamo il problema del ricoprimento con vertici. Dato un grafo $G = (V, E)$, un ricoprimento con vertici del grafo è un insieme $S \subseteq V$ tale che ogni arco in E ha almeno uno dei due vertici in S.

Il problema di ottimizzazione che consideriamo consiste nel trovare un vertex cover che contenga il minimo numero di nodi possible. Una variante più generale, WeightedVC, associa ad ogni nodo i un peso $w_i \geq 0$. Il peso di un insieme di vertici S, è la somma dei pesi dei vertici appartenenti all'insieme, $w(S) = \sum_{i \in S} w_i$. Il problema WeightedVC è quello di trovare un ricoprimento il cui peso sia minimo.

È facile vedere che VertexCover $\leq_P$ WeightedVC: Basta usare $w_i = 1$, per tutti i nodi i, per fare in modo che il peso di un insieme di nodi sia esattamente la sua cardinalità. Quindi, dato che VertexCover è $\mathcal{NP}$-hard, anche WeightedVC è $\mathcal{NP}$-hard. In entrambi i casi possiamo dare un algorimo di approssimazione.

3.2.1 *Un algoritmo di approssimazione per* VertexCover

Per ottenere un algoritmo di approssimazione per VertexCover possiamo utilizzare un approccio semplice: selezionare due nodi qualsiasi uniti da un arco e inserirli nell'insieme ricoprente, cancellare tutti gli archi incidenti sui due nodi selezionati e ripetere il processo fino a che ci sono archi da coprire. Intuitivamente questo approccio funziona bene in quanto dato un arco $e = (u, v)$, un qualsiasi insieme ricoprente, quindi anche quello ottimale, dovrà includere almeno uno fra u e v. Inserendoli entrambi saremo sicuri di avere questa proprietà e inoltre la nostra soluzione non potrà avere più del doppio del minimo numero di nodi necessari per un ricoprimento.

Lo pseudocodice è presentato nell'algoritmo ApproxVertexCover.

Algorithm 9: ApproxVertexCover($G = (V, E)$)

$S = \{\}$
TempSet $= E$
while *TempSet* $\neq \{\}$ **do**
  Sia $e = (u, v)$ un arco in *TempSet*
  $S = S \cup \{u, v\}$
  rimuovi da *TempSet* tutti gli archi incidenti su u o su v
Restituisci S

Vediamo un esempio di esecuzione. Nella Figura 3.2 è schematizzata l'esecuzione dell'algoritmo su uno speicifico grafo, quello a sinistra, con 6 vertici e 8 archi. Nella

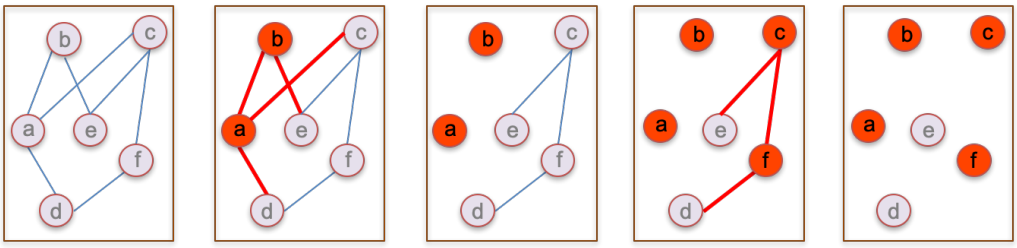

Figura 3.2: Esempio di esecuzione di ApproxVertexCover

prima iterazione l'algoritmo sceglie l'arco (a, b). Quindi inserisce a e b in S e cancella dal grafo tutti gli archi ad essi incidenti. Nella seconda iterazione l'algoritmo sceglie l'arco (c, f). Quindi inserisce c e f in S e cancella dal grafo tutti gli archi ad essi incidenti. A questo punto non ci sono più archi quindi l'algoritmo resituisce $S = \{a, b, c, f\}$.

Si noti che nella seconda iterazione l'algoritmo avrebbe potuto scegliere anche larco (e, c) (o l'arco (d, f), forzando così una terza iterazione che avrebbe portato ad inserire in S tutti i nodi del grafo.

In questo particolare grafo la soluzione ottima è un ricoprimento di 3 nodi, $S^* = \{a, e, f\}$. Dunque in questo caso l'algoritmo approssimato ha fornito una soluzione relativamente vicina all'ottima, anche se avrebbe potuto fornirne una più lontana

che, sempre inq uesto specifico esempio, sarebbe stata anche la soluzione abnale che comprende tutti i nodi del grafo. Ma, al di là della bontà della soluzione, siamo sicuri che qualunque sia il grafo l'algoritmo effettivamente resituisce un ricoprimento? Sì, come provato nel seguente lemma.

Lemma 3.2.1 *L'algoritmo* ApproxVertexCover *restituisce un ricoprimento.*

Dimostrazione. Sia S l'insieme costruito dall'algoritmo. Il fatto che l'algoritmo restituisce S significa anche che *TempSet* è (alla fine) vuoto. Supponiamo per assurdo che S non sia un ricoprimento. Allora esiste un arco $e = (u, v)$ non ricoperto. Ovviamente tale arco è presente nell'insieme *TempSet* all'inizio dell'algoritmo e poichè né u nè v sono stati inseriti in S esso non verrà mai cancellato da *TempSet*. Questo non è possibile perché abbiamo detto che (alla fine) *TempSet* è vuoto. □

L'algoritmo ApproxVertexCover opera in tempo polinomiale nel numero di nodi n del grafo. Infatti il ciclo **while** può durare al massimo un numero di iterazioni pari al numero di archi visto che in ogni iterazione verrà cancellato dall'insieme *TempSet* come minimo l'arco e, se non anche altri archi. Il numero di archi è al massimo n^2. Il tempo necessario alle operazioni da svolgere in ogni singola iterazione dipende da come vengono rappresentati gli insiemi; senza preoccuparci di quale sia l'implementazione più efficiente possiamo sicuramente dire che è possible implementarle in tempo $O(n^2)$ per l'insieme degli archi e $O(n)$ per quelli dei nodi. Quindi l'intero algoritmo, se l'implementazione è ragionevole, non può costare più di $O(n^4)$. Si noti che non ci stiamo preoccupando di quale sia la migliore implementazione, ma solo del fatto che il tempo necessario è polinomiale. Dal punto di vista pratico è ovviamente opportuno individuare la migliore implementazione possibile, cioè quella con tempo di esecuzione più basso. Per quanto detto poc'anzi si ha il seguente lemma.

Lemma 3.2.2 *L'algoritmo* ApproxVertexCover *è efficiente.*

Abbiamo visto che l'algoritmo effettivamente restituisce un ricoprimento e che lo fa in tempo polinomiale. Ci rimane da provare una qualche garanzia sull'approssimazione della soluzione fornita da ApproxVertexCover.

L'intuizione, come abbiamo detto, è che non dovrebbe essere peggiore di 2 volte quella ottima. Infatti si ha il seguente risultato.

Lemma 3.2.3 *L'algoritmo* ApproxVertexCover *è efficiente e 2-approssimato.*

Dimostrazione. Consideriamo una qualsiasi istanza I del problema e sia S la soluzione fornita da ApproxVertexCover su input I e indichiamo con O la soluzione ottima per l'istanza I.

Durante la costruzione di S l'algoritmo ApproxVertexCover ha selezionato un'insieme di archi, uno in ogni iterazione del ciclo **while**. Sia $E(S)$ l'insieme di tali archi.

Osserviamo che la soluzione S contiene un numero di nodi pari esattamente a

$$|S| = 2|E(S)|,$$

in quanto per ogni arco selezionato vengono inseriti i due nodi che l'arco collega.

Inoltre, poichè ogni arco deve essere coperto da almeno un nodo in qualunque insieme ricoprente, quindi anche in O, e poichè gli archi inclusi in $E(S)$ non possono avere nodi in comune (in quanto ogni volta che un arco viene selezionato vengono cancellati tutti gli altri archi incidenti sui due nodi dell'arco), si ha che

$$|E(S)| \leq |O|,$$

cioè la soluzione ottima deve contenere almeno un nodo per ogni arco di $E(S)$.

Pertanto si ha che

$$|S| \leq 2|O|.$$

□

3.2.2 *Un algoritmo di approssimazione per* WeightedVC

Vediamo adesso la versione più generale in cui i vertici hanno dei pesi. Possiamo usare un approcio simile anche se ora dobbiamo tenere conto dei pesi. Pensiamo ai pesi come a dei costi: per coprire ogni arco dovremo *pagare* qualcosa, una frazione del costo totale. In questa prospettiva possiamo pensare ad ogni arco come ad un *agente* disponibile a pagare qualcosa per far coprire l'arco. L'algoritmo, oltre a trovare un ricoprimento, determinerà anche il "prezzo" che tale ricoprimento determina per ogni singolo arco. Diremo che i pagamenti p_e per gli archi incidenti in un vertice i sono *giusti* se la somma non supera il costo del nodo, cioè se $\sum_{e=(i,\cdot)} p_e \leq w_i$.

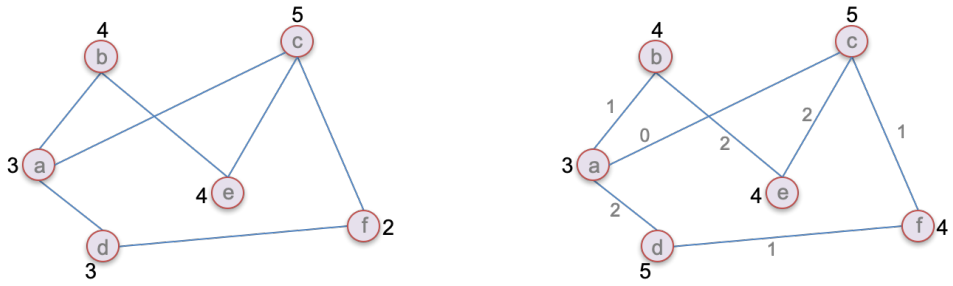

Figura 3.3: Grafo pesato e pagamenti giusto sugli archi

La Figura 3.3 mostra nella parte sinistra un grafo pesato e nella parte destra lo stesso grafo con dei pagamenti sugli archi. Ad esempio l'arco (a, b) ha un pagamento $p_{(a,b)} = 1$ mentre l'arco (b, e) ha un pagamento $p_{(b,e)} = 2$. Il pagamento relativo a un nodo è la somma dei pagamenti degli archi incidenti sul nodo. Ad esempio, per il nodo a il pagamento totale è pari a 3, mentre per il nodo e il pagamento totale è 4.

Si noti come il pagamento di un arco venga inputato ad entrambi i nodi incidenti. Quindi, ad esempio, il pagamento $p_{(a,b)} = 1$ contribuisce sia al pagamento totale relativo al nodo a sia a quello relativo al nodo b.

I pagamenti mostrati nella figura sono giusti in quanto, per ogni singolo nodo, la somma dei pagamenti su tutti gli archi incidenti sul nodo stesso è minore del peso del nodo. Ad esempio per il nodo ha il pagamento totale è 3 ed è uguale al peso del nodo a. Per il nodo f il pagamento totale è 2 ed è minore del peso (4) del nodo stesso.

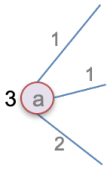

Figura 3.4: Pagamenti non giusti

La Figura 3.13 mostra un esempio di pagamenti non giusti: il pagamento totale per il nodo a è 4 ed è maggiore del peso del nodo che è pari a 3.

Quando la somma dei pagamenti sugli archi incidenti è uguale al peso del nodo, $\sum_{e=(a,\cdot)} p_e \leq w_a$, diremo che il nodo è *saturo*. Ad esempio, il nodo a è saturo mentre il nodo b non lo è.

Lemma 3.2.4 *Per ogni ricoprimento S^*, e ogni pagamento giusto p_e, si ha che $\sum_{e \in E} p_e \leq w(S^*)$.*

DIMOSTRAZIONE. Consideriamo un ricoprimento S^*. Dalla definizione di pagamenti giusti, si ha che $\sum_{e=(i,\cdot)} p_e \leq w_i$, per tutti i nodi $i \in S^*$. Sommando su tutti i nodi di S^* otteniamo

$$\sum_{i \in S^*} \sum_{e=(i,\cdot)} p_e \leq \sum_{i \in S^*} w_i = w(S^*).$$

Osserviamo ora che, poichè S^* è un ricoprimento, ogni arco $e \in E$ appare almeno in una sommatoria del termine a sinistra della precedente diseguaglianza. Potrebbe apparire più di una volta se il ricoprimento contiene entrambi i vertici dell'arco. Pertanto si ha che:

$$\sum_{e \in E} p_e \leq \sum_{i \in S^*} \sum_{e=(i,\cdot)} p_e,$$

dalla quale si conclude che

$$\sum_{e \in E} p_e \leq w(S^*).$$

□

Vediamo ora l'algoritmo approssimato con il quale vogliamo sia trovare un ricoprimento, sia stabilire i pagamenti. L'algoritmo segue un approccio *greedy* rispetto al modo in cui sceglie i pagamenti, cercando cioè di pagare quanto meno possibile.

Algorithm 10: APPROXWEIGHTEDVC(G, w)

$p_e = 0$ per tutti gli archi $e \in E$
while $\exists e = (i, j) \in E$ *tale che né i né j sono saturi* **do**
    Sia e un tale arco
    Incrementa p_e senza violare la proprietà di pagamento giusto.
Sia S l'insieme dei nodi saturi
Restituisci S

Come esempio consideriamo l'esecuzione dell'algoritmo APPROXWEIGHTEDVC sul grafo mostrato nella Figura 3.5. I pagamenti iniziali sono 0 quindi nessun nodo è saturo. Supponiamo che l'algoritmo selezioni l'arco (a, b). Possiamo incrementare il prezzo di tale arco fino a 3, in quanto a quel punto il nodo a diventa saturo. A quel punto l'algoritmo potrebbe selezionare l'arco (b, e) ed aumentare il suo prezzo fino a 1: poiché il peso di b è 4 e su b incidono sia l'arco (a, b) che l'arco (b, e), il pagamento su quest'ultimo può essere al massimo 1 per non violare la proprietà di pagamento giusto. A questo punto anche il nodo b è saturo. Quindi nella successiva iterazione l'algoritmo potrebbe selezionare l'arco (e, c) e portare il pagamento fino a 3 saturando

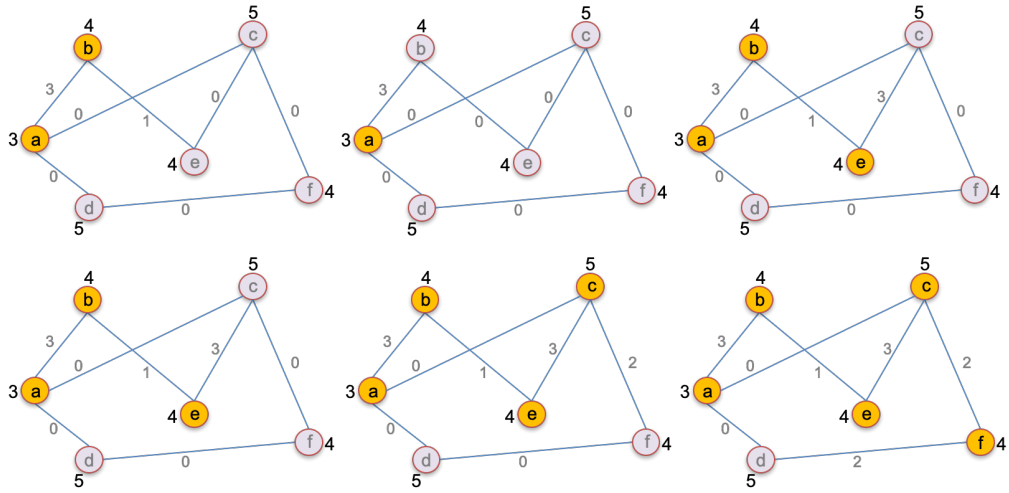

Figura 3.5: Esecuzione ApproxWeightedVC

così il nodo e. Quindi si potrebbe incrementare il pagamento sull'arco (c, f) fino a 2, saturando il nodo c ed infine portare il pagamento sull'arco (f, d) a 2 per saturare il nodo f. A questo punto non ci sono più archi che incidono su due nodi non saturi e quindi l'algoritmo termina restituendo l'insieme de nodi saturi, cioè $\{a, b, c, e, f\}$.

Lemma 3.2.5 *L'insieme S ed i prezzi p_e calcolati dall'algoritmo* ApproxWeightedVC *soddisfano $w(S) \leq 2 \sum_{e \in E} p_e$.*

Dimostrazione. Tutti in nodi di S sono saturi, quindi si ha che $\sum_{e=(i,\cdot)} p_e = w_i$ per tutti i nodi $i \in S$. Pertanto,

$$w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,\cdot)} p_e.$$

Un arco $e = (i, j)$ può apparire al massimo due volte nelle sommatorie del termine a destra (questo succede quanto entrambi i e j fanno parte di S), pertanto si ha che

$$\sum_{i \in S} \sum_{e=(i,\cdot)} p_e \leq 2 \sum_{e \in E} p_e.$$

□

Lemma 3.2.6 *L'insieme S calcolato dall'algoritmo* ApproxWeightedVC *è un ricoprimento con vertici ed il suo costo è al massimo 2 volte il costo di un ricoprimento ottimo.*

Dimostrazione. Provare che S è un ricoprimento è facile in quanto l'algoritmo si ferma solo quando non ci sono archi per cui si può aumentare il prezzo. Quindi dato un arco e, quando l'algoritmo si ferma, almeno uno dei nodi su cui l'arco incide, è saturo e pertanto tale nodo è incluso in S. Quindi ogni arco è ricoperto da almeno un nodo di S.

Il limite sull'approssimazione ottenuta dall'algoritmo deriva facilmente dai Lemmi 3.2.4 e 3.2.5. Sia S^* un ricoprimento ottimale. Dal Lemma 3.2.5 si ha che $w(S) \leq 2\sum_{e \in E} p_e$ mentre dal Lemma 3.2.5 si ha che $\sum_{e \in E} p_e \leq w(S^*)$. Unendo le due diseguaglianze di ha

$$w(S) \leq 2 \sum_{e \in E} p_e \leq 2w(S^*).$$

□

3.2.3 *Programmazione lineare*

Un altro algoritmo approssimato per il problema WeightedVC può essere ottenuto sfruttando una tecnica molto potente usata in ricerca operativa: la *programmazione lineare*. Tale tecnica è l'oggetto di interi corsi per cui non avremo la pretesa di fornire nessuna descrizione approfondita, ma il nostro obiettivo è solo quello di introdurre le idee di base e conoscere quanto basta per applicare la programmazione lineare alla progettazione di algoritmi approssimati.

Nella prossima sezione, utilizzeremo la programmazione lineare per progettare un nuovo algoritmo di approssimazione per WeightedVC.

Per introdurre la programmazione lineare è utile richiamare alcune nozioni di algebra lineare per la risoluzione simultanea di un insieme di equazioni lineari. Usando la notazione matriciale, abbiamo un vettore x di incognite, una matrice A di coefficienti, un vettore b di termini noti. Vogliamo risolvere l'equazione $Ax \geq b$. Ad esempio, se $x = [x_1, x_2]$ è il vettore delle incognite, un sistema di equazioni $Ax \geq b$ è:

$$\begin{aligned} x_1 + 2x_2 &\geq 6 \\ 2x_1 + x_2 &\geq 6 \\ x_1 &\geq 0 \\ x_2 &\geq 0 \end{aligned}$$

dove le ultime 2 equazioni servono a limitare la ricerca a incognite non negative.

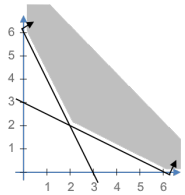

Figura 3.6: Regione ammissibile

Tali equazioni determinano una regione di soluzioni. La Figura 3.6 riporta la regione delle soluzioni dell'esempio appena fatto. Il vettore $b = [6, 6]$ mentre la matrice A è:

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}.$$

In un problema di programmazione lineare abbiamo una funzione obiettivo specificata da un vettore di coefficienti c, che determina la funzione rappresentando una combinazione lineare delle incognite: $c^T x$, dove c^T è il vettore dei coeffienti trasposto, e $c^T x$ è il prodotto di due vettori, che quindi è la combinazione lineare delle incognite espressa dai coefficienti c. Ad esempio se $c = (3/2, 1)$ allora la funzione obiettivo è $3/2x_1 + x_2$.

Un problema di programmazione lineare in forma standard è il seguente.

Data una matrice A di $m \times m$, e dei vettori $b \in \mathbb{R}^m$, $c \in \mathbb{R}^m$, trovare un vettore $x \in \mathbb{R}^m$, per risolvere il seguente problema di ottimizzazione:

$$\min c^T x \text{ vincolato a } Ax \geq b \text{ e } x \geq 0.$$

Per evitare problemi con la rappresentazione dei numeri reali, assumeremo che tutti i numeri coinvolti nella matrice A e nei vettori b e c sono interi.

Nell'esempio fatto in precedenza la soluzione è $x_1 = 2, x_2 = 2$, che corrisponde al punto in cui si intersecano i due vincoli del problema (le due rette). Tale soluzione ci da un valore della funzione obiettivo pari a 5.

Un problema di programmazione lineare può essere anche formulato in versione decisionale nel seguente modo: Data una matrice A, ed i vettori b e c ed un valore γ, esiste un vettore x tale che $x \geq 0$, $Ax \geq b$ e $c^T x \leq \gamma$? Tale problema appartiene a $\mathcal{NP}$. Intuitivamente ciò è credible: infatti dato un vettore x, è facile verificare che il vettore x soddisfa tutti i vincoli imposti dal problema. Ma è veramente facile effettuare la verifica? Il problema però potrebbe nascondersi nei valori delle incognite x_i che essendo dei numeri reali (pososno esserlo anche se tutti i coefficienti sono interi) potrebbe richiedere moltissimi bit per la loro rappresentazione. Un numero irrazionale richiederebbe infiniti bit, non può nemmeno essere rappresentato. Tuttavia, si può dimostrare che se esiste una soluzione allora esiste una soluzione razionale che può essere specificata con un numero polinomiale di bit. Quindi di fatto il problema è in $\mathcal{NP}$ in quanto si può fornire tale soluzione come certificato. Si può dimostrare che oltre ad appartenere a $\mathcal{NP}$ il problema appartiene anche a Co-$\mathcal{NP}$. Per molto tempo, la programmazione lineare, è stato il più famoso esempio di problema appartenente sia $\mathcal{NP}$ che a Co-$\mathcal{NP}$. Poi sono stati scoperti degli algoritmi che lo risolvono in tempo polinomiale.

In pratica però per la risoluzione di un problema di programmazione lineare si utilizza il cosiddetto *metodo del simplesso* che pur non essendo polinomiale funziona molto bene e su istanze di input reali è più efficiente degli algoritmi polinomiali. Solo raramente il metodo del simplesso mostra il suo "carattere esponenziale". Questo fatto ha costituito anche motivo di riflessione sulla definizione di efficienza degli algoritmi: abbiamo definito efficiente un algoritmo polinomiale; ora stiamo presentando un problema per il quale conviene in pratica usare un algoritmo esponenziale.

Il lettore interessato potrà approfondire la conoscenza della programmazione lineare in altri corsi. Per i nostri obiettivi, ci poniamo la seguente domanda: come possiamo utilizzare la programmazione lineare per progettare algoritmi di approssimazione?

3.2.4 WeightedVC *con programmazione lineare*

Ritorniamo al problema della copertura con vertici. Abbiamo un grafo $G = (V, E)$, con dei pesi w_i associati ai vertici $i \in S$ e vogliamo trovare un ricoprimento $S \subseteq V$ di peso totale minimo. Il ricoprimento S deve coprire tutti gli archi, cioè per ogni arco $e = (i, j)$ almeno uno fra i e j deve appartenere a S.

Vogliamo sfruttare la programmazione lineare per risolvere il problema. Useremo una variabile x_i per ogni nodo $i \in V$. Il valore di x_i nella soluzione ci dirà se inserire o meno il nodo i nel ricoprimento: se $x_i = 1$ allora i fa parte del ricoprimento se $x_i = 0$ allora i non fa parte del ricoprimento. Ovviamente dovremo fare in modo che x_i non possa assumere altri valori.

Utilizzeremo le disuguaglianze per codificare il vincolo che i nodi selezionati debbano formare un ricoprimento mentre la funzione obiettivo potrà essere sfruttata per

minimizzare il peso totale. Quindi per ogni arco $(i,j) \in E$, codificheremo il fatto che almeno uno fra i e j deve appartenere al ricoprimento con la disuguaglianza $x_i + x_j \geq 1$. La funzione obiettivo sarà semplicemente data dai pesi che fungeranno da coefficienti $c = [w_1, \ldots, w_n]$. Quindi il problema del ricoprimento con vertici può essere formulato con il seguente problema di programmazione lineare.

$$\min \sum_{i \in V} w_i x_i$$

vincolato da

$$x_i + x_j \geq 1 \text{ per ogni arco } (i,j) \in E$$

$$x_i \in \{0,1\} \text{ per ogni } i \in V.$$

Lemma 3.2.7 *Un insieme di vertici S è un ricoprimento per G se e solo se il vettore x, definito con $x_i = 1$ se $i \in S$ e $x_i = 0$ se $s \notin S$, soddisfa i vincoli del problema di programmazione lineare definito poc'anzi. Inoltre $w(S) = w^T x$.*

DIMOSTRAZIONE. Sia S un ricoprimento e x definito come nell'enunciato. Per ogni arco $(i,j) \in E$ si ha che almeno uno fra i e j appartiente S, quindi almeno uno fra x_i e x_j vale 1 pertanto $x_i + x_j \geq 1$. Ovviamente tutti gli x_i valgono o 0 o 1.

Viceversa sia x un vettore che soddisfa i vincoli e sia $S = \{i | x_i = 1\}$. Poichè $x_i + x_j \geq 1$ per ogni arco $(i,j) \in E$, si ha che S è un copertura di G.

L'uguagliza $w(S) = w^T x$ è immediata in quanto $w^T x = \sum_{i \in S} w_i$. □

Purtroppo il vincolo $x_i \in \{0,1\}$ rende il problema notevolmente più complicato. Infatti si tratta ora di risolvere un problema di programmazione lineare per il quale cerchiamo delle soluzioni intere. Chiameremo questo problema INTEGERPROGRAMMING. Quello che abbiamo mostrato in realtà è che

Lemma 3.2.8 VERTEXCOVER$\leq_P$INTEGERPROGRAMMING.

Il problema INTEGERPROGRAMMING è un problema $\mathcal{NP}$-completo. Per poter procedere allora rinunciamo al vincolo $x_i \in \{0,1\}$, ma richiediamo più semplicimente che $0 \leq x_i \leq 1$. Questo riporta il problema di programmazione lineare in una forma risolvibile efficientemente. Tuttavia ora non sappiamo più se la soluzione al problema LINEARPROGRAMMING fornisce una soluzione buona per il problema originario. Prima di tutto non abbiamo più una ovvia corrispondenza fra la soluzione di LINEARPROGRAMMING e quella di VERTEXCOVER.

Nel seguito utilizzeremo la seguente notazione. Indicheremo con x^*_{IP} la soluzione ottima al problema INTEGERPROGRAMMING e con S^*_{IP} la corrispondente copertura. Indicheremo con x^*_{LP} la soluzione ottima al problema LINEARPROGRAMMING.

Dobbiamo ora trovare un modo per definire una copertura a partire da una soluzione del problema LINEARPROGRAMMING. Se $x_i = 1$ oppure $x_i = 0$ ci troviamo nella stessa situazione di INTEGERPROGRAMMING e possiamo dire che, rispettivamente, $i \in S$ o che $i \notin S$. Ma cosa facciamo se $0 < x_i < 1$? La scelta più ovvia è quella di arrotondare il valore, cioè scegliere il valore intero più vicino. Pertando definiamo la copertura

ponendo $S = \{i | x_i \geq 0.5\}$. L'insieme così definito è effettivamente una copertura? E, se lo è, che possiamo dire sul suo costo?

Data la soluzione ottima x^*_{LP} indicheremo con S_{LP} la corrispondente copertura. Si noti che non usiamo l'asterisco in quanto non sappiamo se la copertura è ottimale. In realtà ancora non sappiamo nemmeno se è una copertura! Iniziamo con l'osservare che effettivamente S_{LP} è una copertura.

Lemma 3.2.9 *L'insieme S_{LP} è una copertura del grafo.*

DIMOSTRAZIONE. Consideriamo un arco $e = (i, j) \in E$. Dalla soluzione del problema di programmazione lineare si ha che $x_i + x_j \geq 1$. Quindi è necessario che o $x_i \geq 1/2$ oppure che $x_j \geq 1/2$. Quindi almeno uno dei due valori sarà arrotondato a 1 e quindi almeno uno dei due vertici i e j sarà incluso in S_{LP}. □

Che possiamo dire del peso di S_{LP}?

Lemma 3.2.10 $w(S_{LP}) \leq 2 \cdot w(S^*_{IP})$.

DIMOSTRAZIONE. Poichè x^*_{IP} è una soluzione a un problema di programmazione lineare intera, si ha che le singole componenti sono 0 o oppure 1 e poichè S^*_{IP} include tutte le componenti corrispondenti a 1 si ha che

$$w(S^*_{IP}) = \sum_{i \in S^*_{IP}} w_i = \sum_{i \in V} w_i x_i = w(x^*_{IP}).$$

D'altra parte poichè il problema LINEARPROGRAMMING ($0 \leq x_i \leq 1$) ha una regione ammissibile che include quella del problema INTEGERPROGRAMMING ($x_i \in \{0, 1\}$), si ha che

$$min_{x_{LP}} \sum_{i \in V} w_i x^{LP}_i \leq min_{x_{IP}} \sum_{i \in V} w_i x^{IP}_i,$$

e quindi che

$$w(x^*_{LP}) \leq w(x^*_{IP}).$$

Pertanto

$$\begin{aligned} w(x^*_{IP}) &\geq w(x^*_{LP}) \\ &= \sum_{i \in V} w_i x^{LP*}_i \\ &\geq \sum_{i \in S_{LP}} w_i x^{LP*}_i \\ &\geq \frac{1}{2} \sum_{i \in S_{LP}} w_i \\ &\geq \frac{1}{2} w(S_{LP}). \end{aligned}$$

Quindi concludiamo che

$$w(S_{LP}) \leq 2w(x^*_{IP}) = 2w(S^*_{IP}).$$

□

3.2.5 *Migliorare il fattore di approssimazione*

Nelle precedenti sezioni abbiamo presentato vari approcci per algoritmi di approssimazione per il problema VertexCover e per la sua generalizzazione WeightedVC. Tutti gli algoritmi presentati hanno un fattore di approssimazione pari a 2. È un caso? È possibile migliorare tale fattore? È l'analisi fatta che può essere migliorata oppure con tali algoritmi non possiamo sperare di fare meglio? In quest'ultimo caso, esistono altri algoritmi con un fattore di approssimazione migliore?

Per stabilire se l'analisi di un algoritmo di approssimazione è la migliore possibile si possono cercare degli esempi di input per i quali l'algoritmo fornisce un'approssimazione che è proprio quella dell'analisi. Questo implicherebbe che l'analisi è stretta e che il fattore trovato è intrinseco nell'algoritmo. Consideriamo ad esempio la seguente tipologia di grafo: grafi bipartiti completi $K_{n/2,n/2}$, con n pari, in cui ognuno degli $n/2$ nodi a sinistra è collegato ad ognuno degli $n/2$ nodi a destra (vedi Figura 3.7).

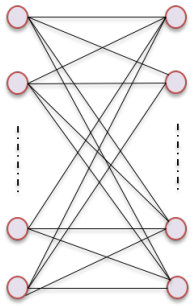

Figura 3.7: Grafo bipartito completo

Come si comporta ApproxVertexCover su questo tipo di grafi? Non è difficile vedere che selezionerà tutti i nodi, quindi la soluzione fornita è composta da n nodi, mentre un ricoprimento ottimale può essere ottenuto selezionando o tutti gli $n/2$ nodi a sinistra o tutti gli $n/2$ nodi a destra. Quindi l'algoritmo, per questi grafi fornisce sempre una soluzione che è 2-approssimata. Dunque l'analisi fatta è stretta, cioè è la migliore in quanto ci sono casi in cui l'algoritmo fornisce soluzioni che hanno un fattore di approssimazione uguale a quello dell'analisi.

É l'algoritmo ApproxVertexCover a non essere sufficientemente buono oppure è il problema che non ammette algoritmi (efficienti) che garantiscano un'approssimazione migliore di 2? Questo è un problema aperto.

3.3 *Load Balancing*

Vediamo adesso un altro problema per il quale l'approccio greedy funziona: il problema di bilanciamento del carico (Load Balancing). Abbiamo m macchine $M_1, \ldots, M_m$ e un insieme di n compiti $C_1, \ldots, C_n$ da svolgere sulle macchine. Ognuno dei compiti necessita di un tempo di esecuzione pari a t_j. Vogliamo assegnare i compiti alle macchine (ogni compito viene assegnato ad una macchina, ogni macchina può svolgere più di un compito, uno dopo l'altro) in modo tale che il carico su ogni macchina sia quanto più bilanciato possibile; cioè la differenza fra il carico massimo e quello minimo deve essere quanto più piccola possibile. Ad esempio nella Figura 3.8 sono mostrati i carichi per 3 macchina; il carico massimo è di 10 unità di tempo e quello minimo di 7, quindi la differenza tra minimo e masismo in questo caso èdi 2 unità di tempo. L'obiettivo è minimizzare tale differenza.

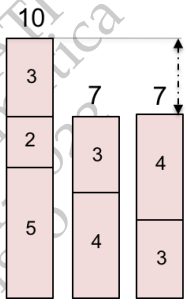

Figura 3.8: Carico per $5, 4, 3, 4, 3, 2, 3$

Più formalmente, dato un assegnamento dei compiti alle macchine, denotiamo con $A(i)$ l'insieme dei compiti assegnati alla macchina i. Poichè la macchina svolge i compiti ad essa assegnati uno dopo l'altro, il tempo di esecuzione necessario alla macchina M_i per completare tutto il lavoro è dunque

$$T_i = \sum_{j \in A(i)} t_j.$$

Faremo riferimento a tale tempo di esecuzione come il "carico" della macchina M_i. Vogliamo minimizzare il carico massimo

$$T = \max_i T_i.$$

Non lo proviamo ma questo problema, apparentemente semplice, è $\mathcal{NP}$-hard. Pertanto presentiamo un algoritmo di approssimazione che si basa sulla tecnica greedy. La scelta greedy che operiamo è la seguente: nell'assegnare un nuovo compito alle macchine scegliamo quella che al momento ha il carico più piccolo.

Algorithm 11: ApproxLoadInOrder

$T_i = 0$ e $A(i) = \{\}$ per tutte le macchine M_i
for $j \leftarrow 1$ **to** n **do**
  Sia M_i una macchina tale che $T_i = \min_k T_k$
  Assegna il compito C_j alla macchina M_i
  $A(i) = A(i) \cup \{j\}$
  $T_i = T_i + t_j$

La Figura 3.8 mostra il risultato dell'esecuzione di questo algoritmo sull'istanza $C = \{5, 4, 3, 4, 3, 2, 3\}$ e $M = \{1, 2, 3\}$. I carichi sono stati processati nell'ordine in cui sono scritti nell'insieme. L'ordine con cui si processano i carichi è importante in quanto la soluzione fornita dall'algoritmo può cambiare se tale ordine cambia.

Processando i compiti in ordine crescente $2, 3, 3, 3, 4, 4, 5$ si ottiene il carico della Figura 3.9, mentre processando compiti in ordine decrescente $5, 4, 4, 3, 3, 3, 2$ si ottiene il carico della Figura 3.10. Tuttavia, nessuna di queste soluzioni è quella ottimale, che è mostrata nella Figura 3.11. Notiamo che le soluzioni non ottimali non si discostano in maniera significativa dall'ottimo. Nel seguito dimostreremo che ciò non è un caso.

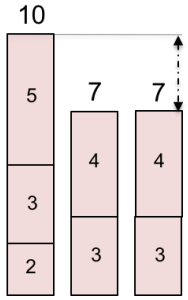

Figura 3.9: Carico per $2, 3, 3, 3, 4, 4, 5$

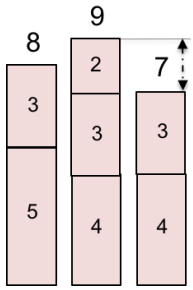

Figura 3.10: Carico per $5, 4, 4, 3, 3, 3, 2$

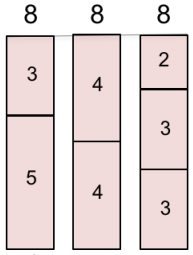

Figura 3.11: Carico ottimale

Analisi. Denotiamo con T il carico massimo che risulta dall'algoritmo. Denotiamo, anche se non lo conosciamo, con T^* il carico massimo ottimale. Il fatto che non conosciamo T^* è ovviamente un problema in quanto per fornire delle garanzie sulla relazione fra T e T^* dovremmo conoscere T^*. Possiamo però individuare un limite inferiore per T^* confrontare tale limite con T: otterremo un'analisi meno precisa nel senso che sarà pessimistica e quindi la garanzia sulla relazione fra T e T^* continuerà a valere.

Chiaramente l'analisi sarà tanto più precisa quanto migliore è il limite che troviamo per T^*. Un limite su T^* è, ad esempio, $T^* \geq t_j$, per un indice j qualsiasi: l'ottimo non può essere inferiore al tempo necessario per eseguire un compito qualsiasi! Tuttavia un tale limite non è molto buono in quanto potrebbe essere estremamente lontano dall'ottimo. Una tale situazione si ha, ad esempio, quando t_j è estremamente piccolo in confronto ai tempi di esecuzione degli altri lavori. In ogni caso tale limitazione ci sarà utile e la possimao scrivere come

$$T^* \geq \max_{1 \leq j \leq n} t_j. \tag{3.1}$$

Un altro limite è

$$T^* \geq \frac{1}{m} \sum_{j=1}^{n} t_j \tag{3.2}$$

perchè se tutte le macchine avessero come carico qualcosa meno di una frazione $\frac{1}{m}$ del totale non si riuscerebbe a svolgere tutti i compiti in quanto ci sono solo m macchine. Anche in questo caso però il limite potrebbe essere molto lontano dall'ottimo se c'è un solo compito il cui tempo di esecuzione è estremamente più grande di tutti gli altri. Anche questo limite ci sarà comunque utile.

Possiamo ora provare il seguente lemma.

Lemma 3.3.1 *L'algoritmo* ApproxLoadInOrder *produce un carico massimo di* $T \leq 2T^*$.

Dimostrazione. Consideriamo una macchina M_i il cui carico è proprio il carico massimo T prodotto dall'algoritmo. Guardiamo l'ultimo compito C_j assegnato a M_i. Quando tale compito è stato assegnato ad M_i, M_i aveva il carico più piccolo. Poichè C_j è l'ultimo compito assegnato a M_i, il carico di M_i prima dell'assegnamento è di $T_i - t_j$. Quindi nel momento in cui C_j è stato assegnato ad M_i, tutte le macchine avevano un carico pari almeno a $T_i - t_i$. Pertanto si ha che

$$\sum_{k=1}^{m} T_k \geq m(T_i - t_j),$$

o, equivalentemente, che

$$T_i - t_j \leq \frac{1}{m} \sum_{k=1}^{m} T_k.$$

D'altra parte $\sum_{k=1}^{m} T_k = \sum_{j=1}^{n} t_j$, quindi si ha che

$$T_i - t_j \leq \frac{1}{m} \sum_{j=1}^{n} t_j$$

e, usando l'equazione (3.2), si ha che

$$T_i - t_j \leq T^*.$$

Ora diamo un limite anche per t_j sfruttando l'equazione (3.1), grazie alla quale possiamo dire che $t_j \leq T^*$. Quindi si ha che

$$T_i = (T_i - t_j) + t_j \leq T^* + T^* = 2T^*.$$

D'altra parte $T_i = T$, quindi $T \leq 2T^*$. □

L'analisi dell'approssimazione fornita dall'algortimo, cioè la prova che la soluzione fornita è al massimo due volte il valore ottimale, è molto buona, nel senso che esistono casi in cui effettivamente l'algoritmo si comporta fornendo delle soluzioni il cui carico massimo è quasi il doppio di quello ottimale; quindi l'analisi fornisce una buona valutazione del reale comportamento dell'algoritmo.

Consideriamo il caso in cui $n = m(m-1)+1$. I primi $m(m-1)$ compiti richiedono un tempo di esecuzione $t_j = 1$. L'ultimo compito invece ha un tempo di esecuzione

molto più grande, $t_n = m$. L'algoritmo ApproxLoadInOrder distribuisce i primi $m(m-1)$ compiti equamente sulle macchine, in modo tale che ogni macchina riceva $m-1$ compiti, e l'ultimo compito su una qualsiasi delle macchine, generando un carico massimo di $2m-1$. La soluzione ottimale in questo caso sarebbe stata quella di mettere il compito grande da solo su una macchina e distribuire equamente gli altri $m(m-1)$ compiti, m su ognuna delle $m-1$ macchine, per un carico massimo di m. Pertanto in questo paricolare esempio, l'approssimazione è di $\frac{2m-1}{m} = 2 - 1/m$, che al crescere di m tende a 2.

Questo caso è veramente il peggiore possibile, nel senso, che con un po' di sforzo si può migliorare la prova del lemma precedente per mostrare che effetivamente il fattore di approssimazione non è 2, ma proprio $2 - 1/m$.

Miglioramento. Il caso peggiore mostrato nell'esempio è dovuto al fatto che il compito più grande è stato processato per ultimo quando ormai gli altri compiti erano già stati assegnati alle macchine. In altre parole i compiti grandi sono "scomodi" da piazzare quando ormai gli altri compiti sono stati già piazzati. Questa osservazione suggerisce che forse conviene assegnare i compiti in ordine di grandezza decrescente.

Algorithm 12: ApproxLoadDec

$T_i = 0$ e $A(i) = \{\}$ per tutte le macchine M_i
Ordina i compiti per t_j decrescente
`/* Quindi ora abbiamo` $t_1 \geq t_2 \geq \ldots \geq t_n$ `*/`
for $j \leftarrow 1$ **to** n **do**
  Sia M_i una macchina tale che $T_i = \min_k T_k$
  Assegna il compito C_j alla macchina M_i
  $A(i) = A(i) \cup \{j\}$
  $T_i = T_i + t_j$

Questa variante garantisce una soluzione che è al massimo 1.5 volte l'ottimo. Per provarlo, iniziamo con l'osservare che

$$\text{Se } n > m \text{ allora } T^* \geq 2t_{m+1}. \tag{3.3}$$

Infatti, consideriamo gli $m+1$ compiti con carico più grande $t_1 \geq t_2 \geq \ldots \geq t_m \geq t_{m+1}$ e vediamo come possono essere assegnati. Poichè ci sono m macchine almeno due di questi compiti devono essere assegnati alla stessa macchina M_j. Poichè entrambi i compiti assegnati a M_j hanno durata $\geq t_{m+1}$, si ha che $T_j \geq 2t_{m+1}$. Poichè il carico massimo è per definizione il massimo fra i carichi delle macchine, si ha l'equazione (3.3).

Lemma 3.3.2 *L'algoritmo* ApproxLoadDec *garantisce che* $T \leq \frac{3}{2}T^*$.

Dimostrazione. Distinguiamo due casi: $n \leq m$ e $n \geq m+1$. Nel primo caso abbiamo un numero di macchine maggiore o uguale al numero di compiti, quindi l'algoritmo assegna al massimo un solo compito ad ogni macchina. Pertanto in questo caso abbiamo $T = T^*$.

Dobbiamo perciò preoccuparci dell'altro caso, in cui $n \geq m+1$. Consideriamo una macchina M_i che ha carico massimo $T_i = T$. Se ad M_i è stato assegnato un solo lavoro, allora si ha $T_i = T^*$, in quanto l'ottimo non può essere più piccolo del tempo di esecuzione di un qualsiasi lavoro. Qundi possiamo limitarci a considerare il caso in cui M_i riceve 2 compiti. Sia C_j l'ultimo compito assegnato a M_i. Deve essere $j \geq m+1$ in quanto l'algoritmo assegnerà i primi m compiti a m macchine diverse. Poichè i compiti vengono assegnati in ordine di durata decrescente, si ha che $t_j \leq t_{m+1}$, e, visto che $n \geq m+1$, dall'equazione (3.3), si ha che $t_{m+1} \leq \frac{1}{2}T^*$, e pertanto otteniamo $t_j \leq \frac{1}{2}T^*$.

A questo punto possiamo procedere esattamente come nella prova del lemma precedente per arrivare a dire che

$$T_i - t_j \leq T^*.$$

Nella prova precedente da questo punto avevemo raggiunto la conclusione sfruttando 3.1. Qui possiamo concludere sfruttando $t_j \leq \frac{1}{2}T^*$ ottenendo

$$T_i \leq T^* + t_j \leq T^* + \frac{1}{2}T^* = \frac{3}{2}T^*.$$

□

Concludiamo con una nota sulla bontà di questa approssimazione. L'analisi che porta al fattore di approssimazione 3/2 non è stretta. Si può infatti dimostrare con un analisi più complessa che l'algoritmo ApproxLoadDec garantisce un fattore di approssimazione pari a 4/3 e tale bound è stretto [15].

3.4 *Problema della selezione del centro*

Cosideriamo il seguente problema: abbiamo n punti e vogliamo selezionare k "centri" per questi n punti. Per "centri" qui intendiamo il fatto che vogliamo che i k punti scelti siano quanto più vicino possibile agli n punti iniziali. Per definire più precisamente il problema, immaginiamo che gli n punti iniziali siano delle città e che i k punti da selezionare siano dei luoghi dove costruire dei centri commerciali. Vogliamo fare in modo che i centri commerciali siano quanto più vicini possibili alle città. Indicheremo con S l'insieme delle città e con C l'insieme dei centri.

Formalmente avremo una funzione distanza che misura la distanza fra i vari luoghi (città e centri commerciali). Ovviamente la distanza rappresenta un costo e come in altri problemi potrebbe non essere la distanza fisica ma misurare altro. Assumeremo comunque che la metrica utilizzata soddisfi le proprietà di una distanza, cioè

- $dist(a,a) = 0$ per ogni $a \in S$.
- simmetria: $dist(a,b) = dist(b,a)$ per tutti $a, b \in S$.
- disuguaglianza triangolare: $dist(a,b) + dist(b,c) \geq dist(a,c)$.

Il nostro obiettivo è quello di far percorrere meno strada possibile per raggiungere i centri commerciali. Più precisamente, il centro commerciale più vicino per una città a è a distanza $dist(a,C) = \min_{c \in C} dist(a,c)$. Quindi il problema è quello di scegliere i

centri C in modo tale da minimizzare il massimo di tali distanze fra tutte le città. Più formalmente diremo che un insieme di centri C forma una r-copertura se ogni città si trova ad una distanza di al massimo r da almeno uno dei centri commerciali, cioè se $dist(a, C) \leq r$ per ogni città $s \in S$. Chiameremo *raggio di copertura di C*, indicato con $r(C)$, il più piccolo valore di r per il quale C è una r-copertura. Quindi $r(C)$ è la massima distanza da una qualsiasi città al centro commerciale più vicino: assumendo che tutti si rechino presso il centro commerciale più vicino, nessuno dovrà percorrere una distanza maggiore di $r(C)$. L'obiettivo del problema è quello di selezionare C in modo tale da minimizzare $r(C)$.

3.4.1 *Algoritmo* ApproxCentriConR

Una prima idea: selezioniamo il primo centro in modo tale che sia la posizione migliore se ci fosse solo questo centro. Per i successivi li posizioniamo ognuno in modo tale da diminuire ogni volta quanto più possibile il raggio di copertura.

Questo approccio è troppo semplice: è facile trovare dei casi in cui trova delle soluzioni molto inefficienti. Ad esempio, consideriamo il caso in cui ci sono solo due città e dobbiamo costruire due centri commerciali. Ovviamente la soluzione migliore è costruire un centro commerciale in ognuna della due città in modo tale da avere $r(C) = 0$. Tuttavia, la scelta greedy proposta seleziona il punto equidistante fra le due città a e b come luogo per il primo centro e dovunque si posizioni il secondo non c'è modo di ridurre il risultante raggio di copertura $r(C) = dist(a, b)/2$.

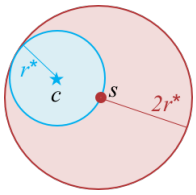

Figura 3.12: Scegliere una città per coprire tutto ciò che è coperto da un centro di una soluzione ottima significa usare un raggio di $2r^*$

Supponiamo adesso di conoscere a priori il valore r^* del raggio di copertura ottimo (nell'esempio precedente tale valore era 0). Indicheremo con C^* un insieme per il quale $r(C^*) = r^*$. Possiamo sfruttare il fatto di sapere che una tale soluzione esiste anche se non la conosciamo. Consideriamo una qualsiasi città $s \in S$. Deve esistere un centro $c \in C^*$ che copre s, cioè tale che $dist(s, c) \leq r^*$. L'idea è che potremmo usare la città s come centro al posto di c, visto che non sappiamo dove c sia, ma sappiamo che dista al massimo r^* da c; quindi scegliendo s possiamo "sbagliare" di al massimo r^*. Per essere sicuri che s copra le stesse città coperte da c dobbiamo quindi accettare che il raggio di copertura possa aumentare da r^* a $2r^*$, come mostrato nella Figura 3.12.

Algorithm 13: ApproxCentriConR

```
S' = S                                        /* S' città da coprire */
C = {}
while S' ≠ {} do
    Seleziona un qualsiasi s ∈ S'
    C = C ∪ {s}
    Cancella tutte le città a distanza minore o uguale a 2r* da s
if |C| ≤ k then
    Restituisci C
else
    Dichiara che non esiste un insieme di k centri con raggio di copertura ≤ r*.
```

Lemma 3.4.1 *Se l'algoritmo* ApproxCentriConR *restituisce un insieme C, tale insieme ha raggio di copertura* $r(C) \leq 2r^*$.

Dimostrazione. Una città viene cancellata dall'insieme S' solo quando viene coperta da un centro che dista al massimo $2r^*$. Pertanto per ogni città s si ha che $dist(s, C) \leq 2r^*$, e quindi $r(C) \leq 2r^*$. □

Rimane da provare che se l'insieme C costruito dall'algoritmo ha più di k centri, e quindi l'algoritmo non restituisce C ma dichiara che non esiste un insieme di al più k centri con raggio di copertura r^*, allora effettivamente non può esistere un insieme di centri con raggio di copertura r^*. In realtà, questa situazione non si può verificare, in quanto sarebbe un assurdo: abbiamo supposto che r^* è il raggio di copertura ottimale per gli insiemi di k centri, quindi deve esistere un insieme con tale raggio di copertura. Proviamolo formalmente.

Lemma 3.4.2 *Supponiamo che l'algoritmo* ApproxCentriConR *selezioni più di k centri. Allora, per ogni insieme C' contenente al massimo k centri, si ha che $r(C') > r^*$.*

Dimostrazione. Sia C l'insieme di centri selezionati dall'algoritmo con $|C| > k$. Per contraddizione, assumiamo che esista un insieme C' con raggio di copertura $r(C') \leq r^*$ e contenente al massimo k centri. Ognuno dei centri $c \in C$ è una delle città di S; per questo motivo chiameremo città-centri i centri di C. Per definizione esiste un centro $c' \in C'$ a distanza al massimo r^* da una qualsiasi città-centro $c \in C$; cioè per ogni $c \in C$, esiste un $c' \in C'$ tale che $dist(c, c') \leq r^*$. Diremo che c' è *vicino* a c.

Osserviamo che quando l'algoritmo ApproxCentriConR seleziona una città-centro cancella dalla lista tutte le città a distanza minore o uguale a $2r^*$. Quindi due qualsiasi città-centri c, d di C sono a distanza più grande di $2r^*$. Pertano se c' è vicino ad uno dei due non può essere vicino all'altro.

Poichè C' ha un raggio di copertura $r(C') \leq r^*$, si ha che ogni città-centro $c \in C$ è vicino ad un centro $c' \in C'$ e, per quanto detto poc'anzi, tutti questi centri di C' sono diversi. Questo significa che $|C'| \geq |C|$ e poichè C contiene più di k centri, si ha che $|C'| > k$. Questa è una contraddizione, quindi l'assunzione che C' sia un insieme di copertura con al massimo k centri e raggio di copertura $r(C') \leq r^*$ è assurda. Tale insieme non può esistere. □

3.4.2 *Algoritmo* ApproxCentriSenzaR

L'algoritmo ApproxCentriConR si basa sulla conoscenza del raggio di copertura ottimo r^*. La conoscenza di tale valore è indispensabile per implementare l'algoritmo. Come facciamo se non lo conosciamo?

L'approccio che abbiamo descritto è utile anche se non conosciamo r^*. Infatti possiamo "indovinarlo" facendo una serie di tentativi. Osserviamo che r^* è maggiore di 0 e minore della distanza massima fra due città r_{max}. Questo ci permette di fare una ricerca binaria, iniziando con il valore $r^* = r_{max}/2$. L'algoritmo può fornire due risposte: un insieme C oppure l'affermazione che non esiste una soluzione con raggio di copertura $r^* = r_{max}/2$. Nel primo caso ripetiamo la ricerca nell'intervallo

sinistro, mentre nel secondo caso la ripetiamo nell'intervallo destro. Pertanto, potremmo iterativamente mantenere due valori $r_0 < r_1$ in modo tale che sappiamo sempre che il valore di r^* è maggiore di r_0 ma anche che abbiamo una soluzione di raggio al massimo $2r_1$. Ad ogni iterazione proviamo con $r^* = (r_0 + r_1)/2$. Dall'output dell'algoritmo capiremo o che una soluzione ottima con tale raggio non esiste e che quindi potremo aggiornare $r_0 = (r_0 + r_1)/2$ oppure che esiste una soluzione con raggio di copertura $2r^* = (r_0 + r_1) < 2r_1$, e quindi potremo aggiornare $r_1 = r^*$. In entrambi i casi abbiamo ristretto il nostro intervallo di ricerca. Potremo fermarci quando i valori di r_0 e r_1 sono sufficientemente vicini, nel qual caso la soluzione di raggio $2r_1$ è abbastanza vicina ad una soluzione 2-approssimata.

La tecnica che abbiamo appena descritto può essere applicata in generale. Per il caso specifico del problema della selezione dei centri, tuttavia, esiste una soluzione più elegante. Di fatto possiamo usare lo stesso approccio dell'algoritmo precedente anche senza conoscere il raggio di copertura ottimale.

L'algorimo precedente, grazie alla conoscenza di r^*, seleziona ripetutamente una delle città come prossimo centro, assicurandosi che sia almeno a distanza $2r^*$ da tutti gli altri centri selezionati. Questo lo fa eliminando dall'insieme S' tutte le città a distanza al massimo $2r^*$ dalla città-centro selezionata. Possiamo ottenere lo stesso effetto anche senza la conoscenza di r^*, semplicemente selezionando la città s che è la più distante da tutte le città-centro già selezionate. Se esistono delle città che distano almeno $2r^*$ da tutte le città-centro già selezionate la città s deve essere una di queste (visto che è quella più distante!). Quindi l'algoritmo diventa il seguente.

Algorithm 14: ApproxCentriSenzaR: Selezione dei centri senza la conoscenza di r^*

```
if k ≥ |S| then
    Restitusci C = S
Seleziona una qualsiasi città b_1 e poni C = {b_1}
while |C| < k do
    Seleziona la città b che massimizza dist(b, C)
    C = C ∪ {b}
Restituisci C
```

Lemma 3.4.3 *L'algoritmo* ApproxCentriSenzaR *restituisce un insieme C di k centri con $r(C) \leq 2r^*$, dove r^* è il raggio di copertura ottimale.*

Dimostrazione. Sia r^* il raggio di copertura ottimale per un insieme di k centri. Sia C l'insieme di k centri restituito dall'algoritmo ApproxCentriSenzaR.

Se $k \geq |S|$ allora $r(C) = 0$ e quindi il lemma è vero. Pertanto consideriamo solo il caso $k < |S|$. Per assurdo, assumiamo che $r(C) > 2r^*$. Questo significa che esiste una città s che dista più di $2r^*$ dal centro ad essa più vicino, cioè tale che $dist(s, C) > 2r^*$.

Sia B il valore di C durante l'esecuzione dell'algoritmo, cioè inizialmente $B = \{b_1\}$, poi alla prima iterazione diventa $B = \{b_1, b_2\}$, alla seconda $B = \{b_1, b_2, b_3\}$, e così via, e sia $b = b_i$ la città selezionata in una generica iterazione dell'algoritmo. Poichè b è selezionata con il criterio di massimizzare la distanza verso tutte le città già presenti in

B, si ha che quando b viene selezionata essa è quella più distante da tutte le città di B, quindi deve necessariamente essere

$$dist(b, B) \geq dist(s, B),$$

altrimenti sarebbe stata selezionata s e non b. Inoltre aggiungendo nuove città a B la distanza $dist(s, B)$ non può che diminuire, quindi

$$dist(s, B) \geq dist(s, C).$$

Ricordando che $dist(s, C) > 2r^*$, si ha

$$dist(b, B) \geq dist(s, B) \geq dist(s, C) > 2r^*,$$

e quindi che

$$dist(b, B) > 2r^*.$$

Ciò è vero per ognuna della k iterazioni dell'algoritmo ApproxCentriSenzaR. Questo significa che ApproxCentriSenzaR è una corretta "implementazione" delle prime k iterazioni dell'algoritmo ApproxCentriConR, in quanto in ogni iterazione seleziona una città che dista più di $2r^*$ da tutte le cità già selezionate.

Poichè $dist(s, C) > 2r^*$, si ha che ApproxCentriConR non terminerà l'esecuzione dopo le prime k iterazioni in quanto s non verrà cancellata da S. Questo significa che ApproxCentriConR concluderà l'esecuzione non fornendo nessun insieme ma dichiarando che tutti gli insiemi di al più k centri hanno un raggio di copertura $> r^*$. Abbiamo già provato che l'algoritmo ApproxCentriConR fornisce correttamente questa risposta, quindi possiamo conlcudere che il raggio di copertura ottimale deve essere $> r^*$. Questa è una contraddizione in quanto abbiamo supposto che r^* è il raggio di copertura ottimale per insiemi con al più k centri, quindi deve esistere un insieme con raggio di copertura r^*.

La contraddizione prova che l'assunzione $r(C) > 2r^*$ è assurda. Dunque $r(C) \leq 2r^*$ □

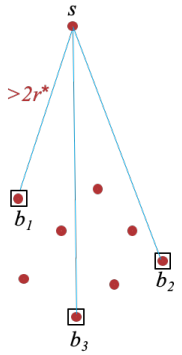

Figura 3.13: L'algoritmo non si ferma perché s non è stata cancellata

3.5 FPTAS: Il problema dello zaino

Ritornando al discorso sulla bontà della soluzione approssimata, vedremo adesso un problema per il quale riusciamo a dare un algoritmo approssimato in cui la soluzione può essere buona quanto si vuole, quindi la si può far avvicinare a piacere all'ottimo. Ovviamente per calcolare una soluzione migliore l'algoritmo necessiterà di più tempo, ma il tempo necessario, sebbene maggiore, rimane comunque polinomiale.

Dunque un obiettivo singificativo è quello di fornire algoritmi approssimati capaci di trovare soluzioni vicine a piacere all'ottimo. Per formalizzare questo aspetto utilizzaremo un parametro ϵ, un numero reale che può essere piccolo a piacere, e vorremo un algoritmo che garantisca una soluzione che non si discosti da quella ottima più di fattore $(1 + \epsilon)$. Si noti come questo leghi ϵ a ρ con la relazione $\epsilon = 1 - \rho$. Detto V il valore della soluzione fornita da un algoritmo di approssimazione, richiederemo che

$$V \leq (1 + \epsilon)OPT \quad \text{per problemi di minimizzazione, e}$$

$$V \geq \frac{OPT}{1+\epsilon} \quad \text{per problemi di massimizzazione.}$$

Chiaramente quanto più piccolo è ϵ (che equivale a quanto più ρ si avvicina a 1) tanto migliore sarà l'approssimazione. Se ad esempio $\epsilon = 0.005$, l'algoritmo fornisce una soluzione peggiore di quella ottima di al massimo lo 0.5%. Inoltre l'algorimo fornirà la soluzione in un tempo che è polinomiale per l'ϵ scelto. È importante qui notare che sebbene ϵ possa essere scelto a piacere, esso è una costante: si sceglie a priori e non può cambiare.

Visto come funzione di ϵ, il tempo di esecuzione potrebbe non essere polinomiale. Quanto più piccolo diventa ϵ tanto più cresce il tempo di esecuzione. In questa ottica ϵ è un parametro e la polinomialità dell'algoritmo di approssimazione va valutata anche in funzione di ϵ; più precisamente per avere un algoritmo pienamente efficiente il tempo di esecuzione deve essere polinomiale, oltre che nella taglia n dell'istanza, anche in $1/\epsilon$ (visto che per trovare soluzioni migliori ϵ deve decrescere).

Tuttavia questo non è un problema dal punto di vista dell'applicazione in casi reali nei quali è spesso sufficiente una buona approssimazione, ad esempio $\epsilon = 0.005$, e quindi non c'è bisogno di rendere ϵ infinitamente piccolo.

Gli algoritmi di approssimazione che operano in questo modo vengono detti *schemi di approssimazione pienamente polinomiali* (FPTAS, Fully Polinomial-Time Approximation Schemes). Chiaramente trovare uno schema di approssimazione pienamente polinomiale è molto desiderabile, ma non sempre possibile. Un problema che ammette un FPTAS è il problema dello zaino.

Nel problema dello zaino, abbiamo n oggetti $1, \ldots, n$ che dobbiamo stipare in uno zaino. Ogni oggetto ha un peso w_i ed un valore v_i. C'è un limite massimo W al peso totale che lo zaino può sostenere. L'obiettivo è quello di inserire un sottoinsieme S di oggetti nello zaino in modo tale da massimizzare il valore totale $\sum_{i \in S} v_i$ e rispettare il vincolo sul peso $\sum_{i \in S} w_i \leq W$.

Algoritmo di programmazione dinamica. Il problema dello zaino, assumendo che i pesi w_i siano degli interi, può essere risolto con un algoritmo di programmazione dinamica che opera in in tempo $O(n^2 W)$, dove W è la capacità dello zaino. Questo algoritmo è quello che, probabilmente, è stato studiato nei corsi di base di algoritmi. Tale algoritmo è *pseudopolinomiale*[2] in quanto pur essendo polinomiale in n e lineare in W, si ha che W può essere esponenziale in n. Quindi l'algoritmo è efficiente solo per valori piccoli di W.

[2] Si ricordi la discussione fatta nella Sezione 1.8.

Descriviamo ora un altro algoritmo, sempre di programmazione dinamica, che risolve il problema dello zaino in tempo $O(n^2 v_{max})$, dove $v_{max} = \max_i\{v_i\}$, assumendo che i valori v_i siano degli interi (per approfondimenti si veda il capitolo su Algoritmi Approssimati di [20]). Ovviamente anche questa variante continua ad essere pseudopolinomiale, ma può essere sfruttata per progettare uno schema di approssimazione pienamente polinomiale.

I sottoproblemi li definiamo in base a due parametri: un indice i e un valore "obiettivo" V: $\text{OPT}(i, V)$ è la più piccola capacità per la quale è possibile inserire nello zaino un sottoinsieme degli oggetti $\{1, 2, \ldots, i\}$ ottenendo un valore di almeno V. Si noti che

avendo definito il problema in questo modo dobbiamo considerare la possibilità che non sia risolvibile: se il valore totale degli oggetti disponibli, $\sum_{i=1}^{n} v_i$, è più piccolo di V allora, indipendentemente dalla capacità dello zaino, non potremo mai raggiungere un valore totale di almeno V. E se per il problema iniziale possiamo ragionevolmente assumere che $\sum_{i=1}^{n} v_i \geq V$ (in altre parole possiamo ignorare i valori di V per cui tale vincolo non è soddisfatto), non vale lo stesso per i sottoproblemi. Per codificare la possibilità che il problema non sia risolvibile useremo il valore di ∞ per la capacità dello zaino.

I sottoproblemi sono definiti per $i = 0, \ldots, n$ e per $V = 0; \ldots, \sum_{j=1}^{i} v_j$. Consideriamo il sottoproblema (i, V). Se $\sum_{i=1}^{n} v_i < V$ allora $OPT(i, V) = \infty$ in quanto non c'è modo di risolverlo. Se, invece, $\sum_{i=1}^{n} v_i \geq V$, allora la soluzione ottima $OPT(i, V)$ è legata a quella di sottoproblemi più piccoli nel seguente modo:

- Se $i \notin OPT(i, V)$ allora $OPT(i, V) = OPT(i-1, V)$
- Se $i \in OPT(i, V)$ allora $OPT(i, V) = w_i + OPT(i-1, \max\{0, V - v_i\})$.

La presenza del $\max\{0, V - v_i\}$ è dovuta al fatto che $V - v_i$ potrebbe essere minore di 0 e ciò non avrebbe senso e quindi lo sostituiamo con 0 (equivalentemente si potrebbe definire $OPT(i, v) = 0$ per tutti i $v < 0$).

Chiaramente si ha che $\sum_{j=1}^{i} v_j \leq n v_{max}$, quindi avremo al massimo $n^2 v_{max}$ sottoproblemi per cui l'algoritmo necessita tempo $O(n^2 v_{max})$. La soluzione al problema originale sarà il massimo valore V per il quale $\mathrm{OPT}(n, V) \leq W$.

Consideriamo un esempio. Abbiamo uno zaino di capacità $W = 60$ e 3 oggetti di peso $w_1 = 40$, $w_2 = 20$ e $w_3 = 50$ e di valore $v_1 = 1$, $v_2 = 1$ e $v_3 = 3$. Decidendo di usare le righe per l'indice i e le colonne per i valori V, la tabella ha dimensione 4×6. Possiamo inizializzarla riempiendo la prima colonna con degli 0 e il resto della prima riga con ∞:

	$V = 0$	1	2	3	4	5
$i = 0$	0	∞	∞	∞	∞	∞
1	0					
2	0					
3	0					

Calcoliamo ora i valori delle soluzioni ottime dei sottoproblemi.

- $(1, 1)$: Quindi $\sum_{j=1}^{i} v_j = 1 \geq V = 1$, quindi il problema è risolvibile.

$$OPT(1, 1) = \min\{OPT(0, 1), 40 + OPT(0, 0)\} = \min\{\infty, 40\} = 40.$$

- $(1, 2)$: Quindi $\sum_{j=1}^{i} v_j = 1 < V = 2$, quindi il problema non è risolvibile. $OPT(1, 2) = \infty$. Chiaramente lo stesso varrà per il resto della riga 1.
- $(2, 1)$: Quindi $\sum_{j=1}^{i} v_j = 2 \geq V = 1$, quindi il problema è risolvibile.

$$OPT(2, 1) = \min\{OPT(1, 1), 20 + OPT(1, 0)\} = \min\{40, 20\} = 20.$$

- $(2,2)$: Quindi $\sum_{j=1}^{i} v_j = 2 \geq V = 2$, quindi il problema è risolvibile.

$$OPT(2,2) = \min\{OPT(1,2), 20 + OPT(1,1)\} = \min\{\infty, 60\} = 60.$$

- $(2,3)$: Quindi $\sum_{j=1}^{i} v_j = 2 < V = 3$, quindi il problema non è risolvibile. $OPT(2,3) = \infty$. Chiaramente lo stesso varrà per il resto della riga 2.

- $(3,1)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 1$, quindi il problema è risolvibile.

$$OPT(3,1) = \min\{OPT(2,1), 50 + OPT(2, \max\{0,-2)\}\} = \min\{20, 50\} = 20.$$

- $(3,2)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 2$, quindi il problema è risolvibile.

$$OPT(3,2) = \min\{OPT(2,2), 50 + OPT(2, \max\{0,-1)\}\} = \min\{60, 50\} = 50.$$

- $(3,3)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 3$, quindi il problema è risolvibile.

$$OPT(3,3) = \min\{OPT(2,3), 50 + OPT(2,0)\} = \min\{\infty, 50\} = 50.$$

- $(3,4)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 4$, quindi il problema è risolvibile.

$$OPT(3,4) = \min\{OPT(2,4), 50 + OPT(2,1)\} = \min\{\infty, 70\} = 70.$$

- $(3,5)$: Quindi $\sum_{j=1}^{i} v_j = 5 \geq V = 5$, quindi il problema è risolvibile.

$$OPT(3,5) = \min\{OPT(2,5), 50 + OPT(2,2)\} = \min\{\infty, 110\} = 110.$$

Per cui la tabella finale è:

	$V=0$	1	2	**3**	4	5
$i=0$	0	∞	∞	∞	∞	∞
1	0	40	∞	∞	∞	∞
2	0	20	60	∞	∞	∞
3	0	20	50	**50**	70	110

Per trovare la soluzione al problema iniziale dobbiamo considerare l'ultima riga e prendere la capacità più grande inferiore a $W = 60$. In questo esempio è 50 e corrisponde al valore $V = 3$. Questo valore è stato ottenuto selezionando il solo terzo oggetto (come succede in qualsiasi algoritmo di programmazione dinamica con un po' di *bookkeeping* si può risalire dal valore nella tabella alla soluzione ottima). Quindi la soluzione ottima è quella di prendere il solo terzo oggetto con peso $w_3 = 50 < W = 60$ e valore $v_3 = 3$.

Algoritmo approssimato. L'idea è quella di sfruttare l'approssimazione per rendere piccolo il valore di v_{max}. Infatti accettando di lavorare su dati approssimati, e quindi di avere una soluzione approssimata, possiamo "scalare" i valori di un fattore b in modo tale da rendere ragionevolmente piccolo il valore di v_{max}. Più nel dettaglio, sia $Z = (v_i, w_i, W)$ il problema di partenza. Consideriamo il problema $\tilde{Z} = (\tilde{v}_i, w_i, W)$,

ottenuto arrotondando ogni valore v_i a $\tilde{v}_i = \lceil \frac{v_i}{b} \rceil b$. Osserviamo che il problema $\tilde{Z}$ è "equivalente" al problema $\overline{Z} = (\overline{v_i}, w_i, W)$, dove $\bar{v}_i = \lceil \frac{v_i}{b} \rceil$. I problemi $\tilde{Z}$ e $\overline{Z}$ sono equivalenti nel senso che possiamo risolvere $\overline{Z}$ e la soluzione ottima fornita per tale problema è ottima anche per $\tilde{Z}$, cambia solo il valore che è scalato di un fattore b. Di contro, la soluzione ottima per $\tilde{Z}$ potrebbe non essere ottima per il problema originario Z, in quanto i valori $\tilde{v}_i$ sono "approssimazioni" dei valori originali v_i. Tuttavia la soluzione ottima per $\tilde{Z}$ è una soluzione approssimata per Z. Scegliendo opportunamento il fattore b potremo ottenere dei valori abbastanza piccoli di $\overline{v_i}$ per fare in modo che $O(n^2 \overline{v}_{max})$ sia sufficientemente piccolo (polinomiale in n) e allo stesso tempo assicurare che l'approssimazione della soluzione per Z sia abbastanza buona.

Iniziamo con l'osservare che:

Lemma 3.5.1 *Per ogni i si ha $v_i \leq \tilde{v}_i \leq v_i + b$.*

DIMOSTRAZIONE. Immediato dalla definizione di $\tilde{v}_i = \lceil \frac{v_i}{b} \rceil b$. □

Inoltre si ha che

Lemma 3.5.2 *Il problema dello zaino sui valori $\tilde{v}_i$ ed il problema dello zaino sui valori $\overline{v_i}$ hanno lo stesso insieme di soluzioni ottime, ed il valore di una soluzione ottima nei due casi differisce per un fattore b.*

DIMOSTRAZIONE. Il lemma è immediato in quanto i pesi non cambiano mentre i valori sono tutti scalati di un fattore b. □

Si noti che il Lemma 3.5.2 ci permette di risolvere il problema $\tilde{Z}$ risolvendo $\overline{Z}$. A questo punto possiamo descrivere l'algoritmo di approssimazione per Z.

Algorithm 15: ZAINOAPPROX($Z = (w, v, \epsilon)$)

$b = \frac{\epsilon}{2n} v_{max}$
Risolvi il problema dello zaino $\bar{Z} = (\bar{v}_i, w_i, W)$; sia S la soluzione
Restituisci l'insieme S

Lemma 3.5.3 *L'insieme S restituito dall'algortimo* ZAINOAPPROX *è tale che $\sum_{i \in S} w_i \leq W$.*

DIMOSTRAZIONE. Questo fatto è immediato in quanto abbiamo arrotondato i valori ma i pesi sono gli stessi quindi la soluzione esatta al problema arrotondato $\tilde{Z}$ è comunque una soluzione ammissibile per il problema originale Z. □

Vediamo ora il tempo di esecuzione.

Lemma 3.5.4 *L'algoritmo* ZAINOAPPROX *calcola una soluzione in tempo polinomiale per $\epsilon > 0$ fissato.*

DIMOSTRAZIONE. Calcolare il valore di b richiede tempo costante. La risoluzione del problema sui valori $\overline{v}$ richiede tempo $O(n^2 \overline{v}_{max})$ dove $\overline{v}_{max} = \max_i \bar{v}_i$.

Ma

$$\bar{v}_{max} = \left\lceil \frac{v_{max}}{b} \right\rceil = \left\lceil \frac{2n v_{max}}{\epsilon v_{max}} \right\rceil = \left\lceil \frac{2n}{\epsilon} \right\rceil = O(n\epsilon^{-1}).$$

Quindi il tempo di esecuzione totale è $O(n^2\tilde{v}_{max}) = O(n^3\epsilon^{-1})$. Dato che ϵ è fissato, cioè è costante, abbiamo un tempo di esecuzione polinomiale in n, più precisamente $O(n^3)$ con una costante nascosta nella notazione O che è tanto più grande quanto più piccolo è ϵ. □

Ci rimane da valutare la bontà della soluzione approssima. Che garanzie abbiamo sull'approssimazione ottenuta?

Lemma 3.5.5 *Sia A la soluzione calcolata dall'algoritmo* ZainoApprox *e sia S^* una soluzione ottima per Z. Si ha che $(1+\epsilon)\sum_{i\in A} v_i \geq \sum_{i\in S^*} v_i$.*

Dimostrazione. Si noti che A è ottima per $\tilde{Z}$ e $\overline{Z}$ (ma potrebbe non esserlo per Z). Poichè A è ottima per $\tilde{Z}$, si ha che

$$\sum_{i\in S^*} \tilde{v}_i \leq \sum_{i\in A} \tilde{v}_i. \tag{3.4}$$

Usando il Lemma 3.5.1 si ha che:

$$\begin{aligned}
\sum_{i\in S^*} v_i &\leq \sum_{i\in S^*} \tilde{v}_i \text{ (Lemma 3.5.1)} && (3.5)\\
&\leq \sum_{i\in A} \tilde{v}_i \text{ (Eq. 3.4)} && (3.6)\\
&\leq \sum_{i\in A} (v_i + b) \text{ (Lemma 3.5.1)} && (3.7)\\
&\leq nb + \sum_{i\in A} v_i. && (3.8)
\end{aligned}$$

Osserviamo che dalla precedente catena di disuguaglianze possiamo estrapolare la disuguaglianza

$$\sum_{i\in A} \tilde{v}_i \leq nb + \sum_{i\in A} v_i$$

dalla quale si ha

$$\sum_{i\in A} v_i \geq \sum_{i\in A} \tilde{v}_i - nb \tag{3.9}$$

Osserviamo che

$$\sum_{i\in A} \tilde{v}_i \geq \tilde{v}_{max}.$$

Questo perchè la soluzione A è ottima per i valori $\tilde{v}_i$ e quindi il suo valore deve essere maggiore del valore che otteniamo scegliendo solo l'oggetto con valore massimo (stiamo assumendo che $w_i \leq W$, per tutti gli oggetti: gli oggetti che hanno un peso maggiore di W possono essere eliminati a priori). Ma $\tilde{v}_{max} = \lceil \frac{v_{max}}{b} \rceil\, b = \lceil \frac{2n}{\epsilon} \rceil b$, quindi

$$\sum_{i\in A} \tilde{v}_i \geq \lceil \frac{2n}{\epsilon} \rceil b.$$

Mettendo insieme la precedente disuguaglianza e la (3.9) si ha che

$$\begin{aligned}\sum_{i\in A} v_i &\geq \sum_{i\in A} \tilde{v}_i - nb \\ &\geq \left\lceil \frac{2n}{\epsilon} \right\rceil b - nb \\ &\geq \frac{2n}{\epsilon} b - nb \\ &= nb\left(\frac{2}{\epsilon} - 1\right)\end{aligned}$$

Una semplice analisi della funzione $2\epsilon^{-1} - 1$ mostra che essa è $\geq \epsilon^{-1}$ per $\epsilon \leq 1$ (assumere che ϵ sia piccolo non è un problema visto che vorremo scegliere ϵ piccolo per otterene buone approssimazioni). Pertanto per $\epsilon \leq 1$ si ha che

$$\sum_{i\in A} v_i \geq \frac{nb}{\epsilon}$$

cioè che

$$nb \leq \epsilon \sum_{i\in A} v_i.$$

Ricordando la relazione 3.8, si ha che

$$\sum_{i\in S^*} v_i \leq \sum_{i\in A} v_i + nb \leq \sum_{i\in A} v_i + \epsilon \sum_{i\in A} v_i \leq (1+\epsilon) \sum_{i\in A} v_i$$

cioè che

$$\sum_{i\in A} v_i \geq \frac{\sum_{i\in S^*} v_i}{1+\epsilon}.$$

□

Dunque per il problema dello zaino esiste uno schema di approssimazione pienamente polinomiale (FPTAS). Un FPTAS è il massimo che possiamo aspettarci da un problema NP-hard (assumendo che $\mathcal{P} \neq \mathcal{NP}$): ci permette di ottenere una soluzione con un'approssimazione buona a piacere usando sempre tempo polinomiale. Sono pochi i problemi che ammettono un FPTAS.

3.6 *Note bibliografiche*

Approfondimenti sugli argomenti presentati in questo capitolo possono essere trovati nel Capitolo 11 di KT2014 [20], nel Capitolo 35 di CLRS2009 [8] e in V2001 [33].

3.7 *Esercizi*

1. Fornire un esempio di grafo per il quale l'algoritmo ApproxVertexCover fornisce sempre una soluzione subottimale.
2. Fornire un esempio di grafo per il quale l'algoritmo ApproxVertexCover fornisce sempre una soluzione ottima.

3. Fornire un algoritmo efficiente (ottimale) per il problema VertexCover nel caso in cui l'input sia un albero.

4. Consideriamo il seguente approccio gredy per il problema VertexCover: inserire nell'insieme ricoprente il nodo con il più alto numero di archi incidenti e quindi cancellare tali archi prima di operare un nuovo inserimento. Fornire un esempio che mostra che tale algoritmo non è 2-approssimato.

5. Il Lemma 1.5.7 suggerisce un algoritmo approssimato per il problema IndependentSet basato sull'algoritmo ApproxVertexCover: usare ApproxVertexCover per trovare un insieme ricoprente approssimato S e restituire come soluzione approssimata del probelma IndependentSet i nodi che non sono in S. Analizzare tale algoritmo e mostrare che potrebbe non avere lo stesso fattore di approssimazione di ApproxVertexCover. Dare una giustificazione e individuare la condizione sotto la quale l'algortimo proposto garantisce una soluzione 2-approssimata.

6. L'analisi dell'algoritmo ApproxWeightedVC è stretta oppure può essere migliorata? Giustificare la risposta.

7. Nell'algoritmo approssimato per il problema WeightedVC basato sulla programmazione lineare, l'insieme ricoprente è definito come $S = \{i | x_i \geq 0.5\}$. Il signor Sergio Aumenta sostiene che usando $S = \{i | x_i \geq 0.75\}$ si ottiene un algoritmo approssimato migliore. La signora Elvira Diminuisce, invece, sostiene che un'approssimazione migliore si ottiene usando $S = \{i | x_i \geq 0.25\}$. Cosa rispondi a Sergio e Elvira?

8. L'algoritmo ApproxLoadInOrder fornisce una soluzione il cui valore è al massimo 2 volte l'ottimo. Ci sono casi in cui può restituire la soluzione ottima? Se sì fornire un esempio, se no argomentare il perchè.

9. L'algoritmo ApproxLoadInOrder fornisce una soluzione il cui valore è al massimo 2 volte l'ottimo. Il valore dipende dall'ordine in cui i lavori sono elencati e quindi allocati. Supponiamo di avere un modo per calcolare la soluzione ottima. Data la soluzione ottima, è possibile riordinare i lavori in modo tale che ApproxLoadInOrder produca la soluzione ottima?

10. Supponi che un corriere di spedizione espressa utilizzi dei furgoni per spedire n pacchi da una sede A ad una sede B. Ogni furgone ha una capacità C e i pacchi da spedire, il cui peso verrà indicato con $w_1, w_2, \ldots, w_n$, vengono caricati sul furgone in ordine di arrivo e quando un forgone non può contenere il prossimo pacco (la somma dei pesi supera C), viene fatto partire e si inizia a caricare un nuovo furgone. Tale algoritmo è un algoritmo di approssimazione.

    (a) Fornire un esempio in cui non viene calcolata la soluzione ottima.

    (b) Provare che l'algoritmo è 2-approssimato.

11. Un tuo amico chimico ti pone il seguente problema. Sta studiando delle molecole $M_1, M_2, \ldots, M_n$ e deve condurre degli esperimenti su ognuna di esse. Questi esperimenti sono costosi e quindi vorrebbe selezionare un sottoinsieme rappresentativo

fatto dal minor numero possibile di molecole sfruttando il fatto che le molecole sono note e che è possibile definire una distanza di similarità fra di esse $d(M_i, M_j)$. L'idea è che se due molecole sono abbastanza simili, cioè $d(M_i, M_j) \leq \delta$, per una fissata soglia δ, allora basta condurre gli esperimenti su una di esse per avere informazioni su entrambe. Aiuta il tuo amico a fornire un algoritmo di approssimazione per selezionare un insieme di molecole sulle quali condurre gli esperimenti. (Possiamo assumere che per ogni molecola M_i ce ne sia almeno un'altra M_j con $d(M_i, M_j) \leq \delta$; se così non fosse per una qualche molecole M_i, tale molecola deve necessariamente essere oggetto degli esperimenti, quindi il problema non si porrebbe e la si potrebbe escludere dall'insieme di input.)

12. Si consideri l'algoritmo ApproxCentriConR. Tale algoritmo seleziona una qualsiasi città fra quelle non ancora cancellate come prossimo centro. Si consideri la seguente variante: si sceglie una qualsiasi città e si considera la circonferenza di raggio r^* in essa centrata; il prossimo centro viene scelto su tale circonferenza. L'algoritmo continua ad essere valido con la stessa approssimazione? Se sì, spiegare il perchè. Se no, dire se può essere fatto funzionare con una approssimazione diversa.